Test Generation Based on Symbolic Specifications

Lars Frantzen[⋆], Jan Tretmans, and Tim A.C. Willemse[⋆⋆]

Nijmegen Institute for Computing and Information Sciences (NIII),
Radboud University Nijmegen – The Netherlands
{lf, tretmans, timw}@cs.ru.nl

Abstract. Classical state-oriented testing approaches are based on simple machine models such as Labelled Transition Systems (LTSs), in which data is represented by concrete values. To implement these theories, data types which have infinite universes have to be cut down to finite variants, which are subsequently enumerated to fit in the model. This leads to an explosion of the state space. Moreover, exploiting the syntactical and/or semantical information of the involved data types is non-trivial after enumeration. To overcome these problems, we lift the family of testing relations $\mathbf{ioco}_{\mathcal{F}}$ to the level of Symbolic Transition Systems (STSs). We present an algorithm based on STSs, which generates and executes tests on-the-fly on a given system. It is sound and complete for the $\mathbf{ioco}_{\mathcal{F}}$ testing relations.

1 Introduction

Testing is an important technique to assess the quality of systems. In testing, experiments are conducted with a System Under Test (SUT) to determine whether it behaves as expected. There are many different kinds of testing. We focus on formal, specification based, black box, functionality testing. This basically means that the SUT can only be observed (and controlled) via its external interfaces. Moreover, a mathematical, unambiguous specification of the causal order between (appropriate) inputs and expected outputs of the SUT is the starting point for the generation and the analysis of the test results.

Several (formal) test generation tools have been developed for specification based, black box testing. Most of these tools use (variations of) state machines or transition systems as the underlying model for test generation. We refer to these types of tools as *state oriented* tools. For an overview of such tools see [2]. A problem, often encountered in such tools is the *state space explosion*, which is

[⋆] Lars Frantzen is supported by the Netherlands Organisation for Scientific Research (NWO) under project: STRESS – Systematic Testing of Realtime Embedded Software Systems.

[⋆⋆] Tim Willemse carried out this work as part of the TANGRAM project under the responsibility of the Embedded Systems Institute. Tangram is partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026.

due to the fact that they use an explicit internal representation for the states of the specification. This is particularly true when the specification uses complex data structures with large or infinite data domains, because each value in the data domain potentially leads to another state. Consequently, many tools can only cope with very restricted data structures with finite domains.

Opposed to state oriented tools are *data type oriented* tools, which are tools tailored to deal with test generation for complicated data structures, such as QuickCheck [3] and Gast [5]. These tools employ the structure of data types to generate test data. However, they lack a built-in concept of state, which makes them less suited to test, e.g., concurrent systems. The way to handle state in such tools is to explicitly define a data structure that represents a state space, but this is not always satisfactory.

The combination of the state oriented and the data type oriented approaches looks promising, and it is exactly this what we investigate in this paper. As our basis we take a state oriented approach to testing, viz. the **ioco** test theory [8]. To the underlying model of Labelled Transition Systems, we add the concept of location variables, and the concept of data, which can be communicated over gates. Both influence the flow of control, thereby allowing us to specify data-dependent behaviour. We refer to these augmented Labelled Transition Systems as *Symbolic Transition Systems* (STSs). We subsequently lift the **ioco** test theory to STSs. As a result, we obtain a sound and complete test derivation algorithm from specifications expressed as STSs.

The test derivation algorithm for STSs allows to treat data symbolically. Rather than elaborating our approach for a specific data formalism, data types are treated as sets of values (algebras) and first order formulas are used to specify values or predicates. This allows to combine STSs with any formalism of choice (with corresponding test tools) for the specification and manipulation of data. This is further elaborated into a tractable algorithm.

From a theoretical point of view, it is also interesting to give an algorithm which generates *symbolic test cases* (STCs). This requires a purely symbolic version of the $\mathbf{ioco}_{\mathcal{F}}$ relations. This is depicted in Fig. 1. The front triangle

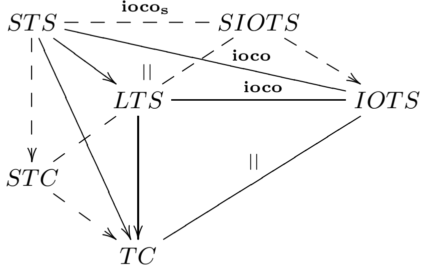


Fig. 1. Classical **ioco** test theory and symbolic **ioco** test theory

represents the classical **ioco** test theory, as presented in [8]. Test cases (TC) are generated out of a specification LTS, and subsequently executed ($||$) on an SUT, assumed to be modelled by an IOTS. The rear triangle consists of a purely symbolic test theory. In this paper, we concentrate on the relation between STSs, LTSs and IOTSs, and on the generation and execution of test cases, i.e. the relation between STSs and TCs. Elaborating on the dashed lines and the corresponding models is another line of research we are pursuing.

Related Work. The idea of combining data type oriented and state oriented approaches is not entirely new in testing. We mention a few noteworthy approaches.

The approach which comes closest to ours is the one described in [7]. There, Input-Output Symbolic Transition Systems (IOSTSs) are used, which are very similar to our STSs. The conformance relation they use corresponds to **ioconf** $=$ $\mathbf{ioco}_{traces(\mathcal{L})}$, but they do not deal with quiescence. In [7] test purposes are chosen as a way to tackle the state space explosion problem. These are used to compute a subgraph of the IOSTS representing a specific issue of interest. Such test purposes are again (special) IOSTSs. The result is a test case which is still symbolic in the sense that it is a deterministic IOSTS with special states *Pass*, *Fail* and *Inconclusive*. The verdict *Inconclusive* is necessary to judge a behaviour which conforms to a given specification, but does not satisfy the given test purpose. Our approach does not rely on test purposes, even though the set $\mathcal{F}$ which identifies the relation $\mathbf{ioco}_{\mathcal{F}}$ can be seen as some form of test purpose.

The data-type oriented GAST tool [5] was recently extended in [6] to deal with specifications given as (possibly nondeterministic) Extended Finite State Machines (EFSMs). Such EFSMs are also symbolic specifications, but in some senses more restrictive than STSs or IOSTSs. GAST basically implements a generic algorithm to enumerate the elements of an arbitrary algebraic data type. Such a type can be an input value, but also a whole path through the EFSM. Since the list of all elements of a recursive type is infinitely long, lazy evaluation is employed to generate only the fraction of this list that is actually needed. The elements are generated in increasing size, both the executed paths and the input values. GAST can be used to execute the generated tests on an SUT in an on-the-fly manner.

Overview. This paper is structured as follows. In Sect. 2 we briefly repeat notions from first order logic. The **ioco** test theory is summarised in Sect. 3. The framework of Symbolic Transition Systems is introduced in Sect. 4. We present an on-the-fly implementation for generating and executing test cases for Symbolic Transition Systems in Sect. 5. We finish with conclusions and future extensions in Sect. 6.

2 First Order Logic

We use basic concepts from first order logic as our framework for dealing with data. For a general introduction into logic we refer to [4]. From hereon we assume a first order structure as given, i.e.:

– A logical signature $\mathfrak{S} = (F,\ P)$ with
  - F is a set of *function symbols.* Each $f{\in}F$ has a corresponding arity $n{\in}\mathbb{N}$. If $n = 0$ we call f a *constant.*
  - P is a set of *predicate symbols.* Each $p{\in}P$ has a corresponding arity $n{>}0$.

– A model $\mathfrak{M} = (\mathfrak{U},\ (f_{\mathfrak{M}})_{f\in F},\ (p_{\mathfrak{M}})_{p\in P})$ with
  - $\mathfrak{U}$ being a nonempty set called *universe.*
  - For all $f{\in}F$ with arity n, $f_{\mathfrak{M}}$ is a function of type $\mathfrak{U}^n{\to}\mathfrak{U}$.
  - For every $p{\in}P$ with arity n we have $p_{\mathfrak{M}} \subseteq \mathfrak{U}^n$.

For simplicity, and without loss of generality we restrict to one-sorted signatures. Let $\mathfrak{X}$ be a set of *variables. Terms* over X, denoted $\mathfrak{T}(X)$, are built from function symbols F and variables $X \subseteq \mathfrak{X}$. We write $\mathsf{var}(t)$ to denote the set of variables appearing in a term t. Terms $t{\in}\mathfrak{T}(\emptyset)$ are called *ground terms.*

Example 1. Assume we have $X = \{x, y\}$. Let $\mathfrak{S} = (F,\ P)$ be given by $F = \{\mathtt{zero}, \mathtt{succ}, \mathtt{add}\}$ (with arities $0, 1$ and 2, resp.), and $P = \{\mathtt{leq}\}$ (with arity 2). An obvious model for this signature is the natural numbers with 0, *successor*, *addition* and the less-or-equal predicate; any other model that sticks to the given arities is fine too. Terms are, e.g. x, $\mathtt{succ}(x)$ and $\mathtt{add}(\mathtt{succ}(x), y)$. Ground terms are, e.g. $\mathtt{zero}$ and $\mathtt{add}(\mathtt{zero}, \mathtt{succ}(\mathtt{zero}))$. □

A *term-mapping* is a function $\sigma{:}\mathfrak{X} \to \mathfrak{T}(\mathfrak{X})$. The term-mapping id, referred to as the *identity mapping*, is defined as $\mathsf{id}(x) = x$ for all $x{\in}\mathfrak{X}$. We use the following notation. For sets X, Y with $X \cup Y \subseteq \mathfrak{X}$, we write $\mathfrak{T}(Y)^X$ for the set of term-mappings that assign to each variable $x{\in}X$ a term $t{\in}\mathfrak{T}(Y)$, and to each variable $x \notin X$ the term x. Given a term-mapping $\sigma{\in}\mathfrak{T}(Y)^X$ we overload the var-notation as follows: $\mathsf{var}(\sigma) =_{def} \bigcup_{x\in X} \mathsf{var}(\sigma(x))$.

The set of free variables of a first order formula φ is denoted $\mathsf{free}(\varphi)$; the set of bound variables is denoted $\mathsf{bound}(\varphi)$. The set of first order formulas φ over $X \subseteq \mathfrak{X}$ is denoted $\mathfrak{F}(X)$; we have $\mathsf{free}(\varphi) \cup \mathsf{bound}(\varphi) \subseteq X$. A tautology is represented by $\top$. The *existential closure* of a formula φ, denoted $\overline{\exists}\varphi$, is defined as $\overline{\exists}\varphi =_{def} \exists x_1 \exists x_2 \ldots \exists x_n : \varphi$ with $\{x_1, \ldots, x_n\} = \mathsf{free}(\varphi)$.

Given a term-mapping σ and a formula φ, the *substitution* of $\sigma(x)$ for $x{\in}\mathsf{free}(\varphi)$ in φ is denoted $\varphi[\sigma]$. Substitutions are side-effect free, i.e. they do not add bound variables. This is achieved using α-renaming. The substitution of terms $\sigma(x)$ for variables $x{\in}\,\mathsf{var}(t)$, in a term t using a term-mapping σ, is denoted $t[\sigma]$.

Example 2. An example of a term mapping for $X = \{x, y\}$ is $\sigma = \{x \mapsto \mathtt{succ}(y),\ y \mapsto \mathtt{zero}\}{\in}\mathfrak{T}(X)^X$, with $\mathsf{var}(\sigma) = \{y\}$. The existential closure of the formula $\varphi = \forall y : \mathtt{leq}(x, y)$ with $\mathsf{bound}(\varphi) = \{y\}$ and $\mathsf{free}(\varphi) = \{x\}$ is $\overline{\exists}\varphi = \exists x \forall y : \mathtt{leq}(x, y)$. The substitution of σ in φ is not side-effect free, but can be achieved by renaming variable y to z, i.e. $\varphi[\sigma] = \forall z : \mathtt{leq}(\mathtt{succ}(y), z)$. □

A *valuation* ϑ is a function $\vartheta{:}\mathfrak{X} \to \mathfrak{U}$. We denote the set of all valuations as $\mathfrak{U}^{\mathfrak{X}} =_{def} \{\vartheta{:}\mathfrak{X} \to \mathfrak{U} \mid \vartheta \text{ is a valuation of } \mathfrak{X}\}$. For a given $X \subseteq \mathfrak{X}$ we write $\vartheta{\in}\mathfrak{U}^X$ when only the values of the variables in X are of interest. For all the other variables $y{\in}\mathfrak{X} \setminus X$ we set $\vartheta(y) = *$, where $*$ is an arbitrary element of set $\mathfrak{U}$.

Having two valuations $\vartheta \in \mathfrak{U}^X$ and $\varsigma \in \mathfrak{U}^Y$ with $X \cap Y = \emptyset$, their union is defined as:

$$(\vartheta \cup \varsigma)(x) =_{def} \begin{cases} \vartheta(x) & \text{if } x \in X \\ \varsigma(x) & \text{if } x \in Y \\ * & \text{otherwise} \end{cases}$$

The *satisfaction* of a formula φ w.r.t. a given valuation ϑ is denoted $\vartheta \models \varphi$. When $\mathsf{free}(\varphi) = \emptyset$ we write $\mathfrak{M} \models \psi$ because the satisfaction is independent of a concrete valuation.

The extension to evaluate whole terms based on a valuation ϑ is called a *term-evaluation* and denoted $\vartheta_{\mathsf{eval}}{:}\mathfrak{T}(\mathfrak{X}) \rightarrow \mathfrak{U}$. The evaluation of ground terms is denoted $\mathsf{eval}{:}\mathfrak{T}(\emptyset) \rightarrow \mathfrak{U}$.

To ease notation, we often treat a tuple $\langle x_1, \ldots, x_n \rangle \in A_1 \times \cdots \times A_n$ as the set $\{x_1, \ldots, x_n\}$. We denote the composition of functions $f{:}B {\rightarrow} C$ and $g{:}A {\rightarrow} B$ as $f \circ g$.

Example 3. Assuming the standard model for natural numbers as given in example 1, an example valuation is $\vartheta = \{x \mapsto 24,\ y \mapsto 7\} \in \mathfrak{U}^{\{x,y\}}$. For the formula φ of example 2, the valuation ϑ and the standard model for natural numbers we find $\vartheta \not\models \varphi$ and $\mathfrak{M} \models \overline{\exists}\varphi$ and we get $\vartheta_{\mathsf{eval}}(\mathtt{add}(x, \mathtt{succ}(y))) = 32$. □

Our example of a logical structure for natural numbers shows that many, even infinite ground terms may evaluate to the same value, e.g. the ground terms `zero` and `add(zero, zero)` both evaluate to 0. We assume we have a unique ground term representative for every value to facilitate the bidirectional translation.

3 Testing Labelled Transition Systems

We briefly review the **ioco**$_{\mathcal{F}}$ test theory on which this paper is based. For a more detailed overview, we refer to [8]. The semantical model we use to model reactive systems is based on *Labelled Transition Systems* (LTSs).

Definition 1. *A* Labelled Transition System *is a tuple* $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$, *where*

- S *is a (possibly infinite) set of* states.
- $s_0 \in S$ *is the* initial state.
- Σ *is a (possibly infinite) set of* action labels. *The special action label* $\tau \notin \Sigma$ *denotes an* unobservable *action. In contrast, all other actions are* observable. *We write* Σ_τ *to denote the set* $\Sigma \cup \{\tau\}$.
- $\rightarrow \subseteq S \times \Sigma_\tau \times S$ *is the* transition relation. *When* $(s, \mu, s') \in \rightarrow$ *we write* $s \xrightarrow{\mu} s'$.

We often identify an LTS $\mathcal{L}$ *with its initial state* s_0.

Unobservable actions can be used to model events that cannot be seen by an observer of a system. The generalised transition relation $\Longrightarrow \subseteq S \times \Sigma^* \times S$ captures this phenomenon: it abstracts from τ actions preceding, in-between and following a (possibly empty) sequence of observable actions. Given an LTS

Table 1. Deduction rules for generalised transitions

$$s \overset{\epsilon}{\Longrightarrow} s \qquad \frac{s \overset{\sigma}{\Longrightarrow} s'' \quad s'' \xrightarrow{\tau} s'}{s \overset{\sigma}{\Longrightarrow} s'} \qquad \frac{s \overset{\sigma}{\Longrightarrow} s'' \quad s'' \xrightarrow{\mu} s' \quad \mu \neq \tau}{s \overset{\sigma\mu}{\Longrightarrow} s'}$$

$\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$, this relation is defined by the deduction rules of Table 1. We define two operations on LTSs. Given an LTS $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$ and a (possibly new) action μ. The *action prefix* $\mu;\mathcal{L}$ is defined as

$$\mu;\mathcal{L} =_{def} \langle S \cup \{s\}, s, \Sigma \cup \{\mu\}, \rightarrow \cup \{s \xrightarrow{\mu} s_0\} \rangle \tag{1}$$

with $s \notin S$ being a fresh state. For a set of LTSs $\overline{\mathcal{L}} = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$ with $n \geq 0$ of the form $\mathcal{L}_i = \langle S_i, s_{0i}, \Sigma_i, \rightarrow_i \rangle$, we define the *alternative composition* of all LTSs $\mathcal{L}_i$, denoted $\sum(\overline{\mathcal{L}})$, as follows:

$$\sum(\overline{\mathcal{L}}) =_{def} \langle \bigcup_{i \leq n} S_i \cup \{s\}, s, \bigcup_{i \leq n} \Sigma_i, \bigcup_{i \leq n} (\rightarrow_i \cup \{s \xrightarrow{\mu} s' \mid s_{0i} \xrightarrow{\mu} s'\}) \rangle \tag{2}$$

with $s \notin \bigcup_{i \leq n} S_i$ being a fresh state. The operator $\sum$ is associative and commutative. We sometimes write $\mathcal{L}_1 + \mathcal{L}_2$ instead of $\sum\{\mathcal{L}_1, \mathcal{L}_2\}$.

3.1 The Test Relation $\mathbf{ioco}_{\mathcal{F}}$

We introduce the following shorthand notation. For a $\mu \in \Sigma_\tau$ we write $s \xrightarrow{\mu}$ when there is a state s' such that $s \xrightarrow{\mu} s'$, and, likewise, given a $\sigma \in \Sigma^*$ we write $s \overset{\sigma}{\Longrightarrow}$ when there is a state s' such that $s \overset{\sigma}{\Longrightarrow} s'$.

Definition 2. *Let $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$ be an LTS and let $s \in S$.*

1. *$init(s) =_{def} \{ \mu \in \Sigma_\tau \mid s \xrightarrow{\mu} \}$.*
2. *$traces(s) =_{def} \{ \sigma \in \Sigma^* \mid s \overset{\sigma}{\Longrightarrow} \}$.*
3. *$\mathcal{L}$ has* finite behaviour *if all $\sigma \in traces(s_0)$ satisfy $|\sigma| < n$ for some $n \in \mathbb{N}$.*
4. *$\mathcal{L}$ is* deterministic *if for all $\sigma \in \Sigma^*$, $|\{s' \mid s_0 \overset{\sigma}{\Longrightarrow} s'\}| \leq 1$.*

We assume that implementations of a reactive system can be given as an *input-output transition system* (IOTSs). An IOTS is an LTS in which the set of action labels Σ is partitioned in a set of *input actions* Σ_I and a set of *output actions* Σ_U, and for which it is assumed that all input actions are enabled in all states.

Definition 3. *Let $\mathcal{L} = \langle S, s_0, \Sigma_I \cup \Sigma_U, \rightarrow \rangle$ be an LTS. A state $s \in S$ is* quiescent*, denoted by $\delta(s)$, if $\forall \mu \in \Sigma_U \cup \{\tau\} : \; s \overset{\mu}{\not\rightarrow}$.*

Let δ be a special action label, not part of any action label set. For a given set of action labels Σ, we abbreviate $\Sigma \cup \{\delta\}$ with Σ_δ. The suspension transitions $\Longrightarrow_\delta \subseteq S \times \Sigma_\delta^* \times S$ are given by the deduction rules of Table 2. The set of all *suspension traces* of $\mathcal{L}$ is denoted $Straces(\mathcal{L}) = \{\sigma \in \Sigma_\delta^* \mid \mathcal{L} \overset{\sigma}{\Longrightarrow}_\delta\}$.

Table 2. Deduction rules for suspension transitions

$\dfrac{s \overset{\sigma}{\Longrightarrow} s'}{s \overset{\sigma}{\Longrightarrow}_\delta s'}$	$\dfrac{\delta(s)}{s \overset{\delta}{\Longrightarrow}_\delta s}$	$\dfrac{s \overset{\sigma}{\Longrightarrow}_\delta s'' \qquad s'' \overset{\upsilon}{\Longrightarrow}_\delta s'}{s \overset{\sigma\upsilon}{\Longrightarrow}_\delta s'}$

Definition 4. *Let* $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow\rangle$ *be an LTS, let* $s \in S$ *be a state and let* $\sigma \in \Sigma_\delta^*$ *be a suspension trace. We define* $s\,\textbf{after}\,\sigma =_{def} \{\ s' \mid s \overset{\sigma}{\Longrightarrow}_\delta s'\ \}$*. We overload this notation as follows:* $\mathcal{C}\,\textbf{after}\,\sigma =_{def} \bigcup_{s\in C} s\,\textbf{after}\,\sigma$*, where* $\mathcal{C} \subseteq S$*.*

The set of *observations* that can be made in a specific state s is given by the set of all output actions that are possible from that state. When no output action is possible the only observation that can be made is quiescence.

Definition 5. *Let* $\mathcal{L} = \langle S, s_0, \Sigma_I \cup \Sigma_U, \rightarrow\rangle$ *be an LTS and let* $s \in S$ *be a state. We define* $\textbf{out}(s) =_{def} \{\delta\}$ *if* $\delta(s)$ *and otherwise* $\textbf{out}(s) =_{def} \{\mu \in \Sigma_U \mid s \xrightarrow{\mu}\}$*. We overload this notation as follows:* $\textbf{out}(\mathcal{C}) =_{def} \bigcup_{s\in C} \textbf{out}(s)$*, where* $\mathcal{C} \subseteq S$*.*

Next, we define the conformance relation $\textbf{ioco}_\mathcal{F}$.

Definition 6. *Let* $\mathcal{F} \subseteq \mathit{Straces}(\mathcal{L})$ *be a subset of suspension traces of a* specification $\mathcal{L}$*. When a (physical) implementation (given as an IOTS)* $\mathcal{P}$ *is* $\textbf{ioco}_\mathcal{F}$*-conform to* $\mathcal{L}$ *we write* $\mathcal{P}\,\textbf{ioco}_\mathcal{F}\,\mathcal{L}$*, where:*

$$\mathcal{P}\,\textbf{ioco}_\mathcal{F}\,\mathcal{L} \textit{ iff } \forall \sigma \in \mathcal{F} : \textbf{out}(\,\mathcal{P}\,\textbf{after}\,\sigma\,) \subseteq \textbf{out}(\,\mathcal{L}\,\textbf{after}\,\sigma\,) \tag{3}$$

3.2 Testing for $\textbf{ioco}_\mathcal{F}$

A *test case* is a special LTS, which is executed on a given SUT. It has a tree-like structure with leaves **pass** and **fail**. To formally differentiate between observed quiescence and specified quiescence, we use θ instead of δ in the test cases, representing observed quiescence.

Definition 7. *A* test case *is an LTS* $t = \langle S, s_0, \Sigma_I \cup \Sigma_U \cup \{\theta\}, \rightarrow\rangle$*, satisfying:*

– *t is deterministic and has finite behaviour.*
– $\{\textbf{pass}, \textbf{fail}\} \subseteq S$ *are terminal states satisfying* $\mathit{init}(\textbf{pass}) = \mathit{init}(\textbf{fail}) = \emptyset$*.*
– *for any state* $s \in S \backslash \{\textbf{pass}, \textbf{fail}\}$ *either* $\mathit{init}(s) = \{\mu\}$ *for some input* $\mu \in \Sigma_I$ *or* $\mathit{init}(s) = \Sigma_U \cup \{\theta\}$*.*

Test cases are executed simultaneously with implementations. While their inputs and outputs must be executed synchronously, quiescence is synchronised with the θ action of a test case and internal actions of the implementation are executed autonomously. Let $\mathcal{P} = \langle S, s_0, \Sigma_I \cup \Sigma_U, \rightarrow_\mathcal{P}\rangle$ be an IOTS and $t = \langle T, t_0, \Sigma_I \cup \Sigma_U \cup \{\theta\}, \rightarrow_t\rangle$ a test case. The simultaneous execution of t and $\mathcal{P}$ is defined by the LTS $t]|\mathcal{P} = \{T \times S, (t_0, s_0), \Sigma_I \cup \Sigma_U \cup \{\theta\}, \rightarrow\rangle$, where $\rightarrow$ is defined by the rules of Table 3. We say that an implementation $\mathcal{P}$ *passes a test suite* T (i.e. a set of test cases) iff for all its test cases, no test run leads to the verdict **fail**.

Table 3. Deduction rules for synchronous execution

$$\frac{\mathcal{P} \xrightarrow{\tau}_{\mathcal{P}} \mathcal{P}'}{t\rceil\!|\,\mathcal{P} \xrightarrow{\tau} t\rceil\!|\,\mathcal{P}'} \qquad \frac{t \xrightarrow{\mu}_t t' \qquad \mathcal{P} \xrightarrow{\mu}_{\mathcal{P}} \mathcal{P}' \qquad \mu\in\Sigma_I \cup \Sigma_U}{t\rceil\!|\,\mathcal{P} \xrightarrow{\mu} t'\rceil\!|\,\mathcal{P}'} \qquad \frac{t \xrightarrow{\theta}_t t' \qquad \delta(\mathcal{P})}{t\rceil\!|\,\mathcal{P} \xrightarrow{\theta} t'\rceil\!|\,\mathcal{P}}$$

$$\mathcal{P} \ \mathbf{passes}\ T \text{ iff } \forall t\in T : \forall\sigma\in(\Sigma_I \cup \Sigma_U \cup \{\theta\})^* : \forall\mathcal{P}' : t\rceil\!|\,\mathcal{P} \overset{\sigma}{\not\Longrightarrow} \mathbf{fail}\rceil\!|\,\mathcal{P}' \qquad (4)$$

In [8] an algorithm is presented which, given a specification LTS $\mathcal{L}$ and a set $\mathcal{F} \subseteq \mathit{Straces}(\mathcal{L})$, produces test cases for $\mathbf{ioco}_{\mathcal{F}}$. We recapitulate the algorithm, expressed in a slightly simpler way.

Definition 8. *Let $\mathcal{L} = \langle S, s_0, \Sigma_I \cup \Sigma_U, \rightarrow\rangle$ be an LTS and let $\mathcal{F} \subseteq \mathit{Straces}(\mathcal{L})$. Let $\mathcal{C} \subseteq S$ be a non-empty set of states, initially $\mathcal{C} = \{s_0\}$. We use two special LTSs which contain the terminal states* **pass** *and* **fail***:*

$$\mathbf{pass} =_{def} \langle\{\mathbf{pass}\}, \mathbf{pass}, \emptyset, \emptyset\rangle$$
$$\mathbf{fail} =_{def} \langle\{\mathbf{fail}\}, \mathbf{fail}, \emptyset, \emptyset\rangle$$

A test case t is obtained from $\mathcal{C}$ by a finite number of recursive applications of one of the following three nondeterministic choices:

– $t := \mathbf{pass}$
The single-state test case **pass** *is always a sound test case. It stops the recursion and terminates the test case.*

– $t := \mu\ ;\ t'$
where $\mu\in\Sigma_I$ and $\mathcal{C}\,\mathbf{after}\,\mu \neq \emptyset$. We obtain t' by recursively applying the algorithm for $\mathcal{C}' = \mathcal{C}\,\mathbf{after}\,\mu$ and $\mathcal{F}' = \{\sigma\in\Sigma_\delta^* \mid \mu\cdot\sigma\in\mathcal{F}\}$.

– $$\begin{aligned} t := &\textstyle\sum\{\mu; \mathbf{fail} \mid \epsilon\in\mathcal{F} \text{ and } ((\mu\in\Sigma_U, \mu\notin\mathbf{out}(\mathcal{C})) \text{ or } (\mu=\theta, \delta\notin\mathbf{out}(\mathcal{C})))\}\\ &+ \textstyle\sum\{\mu; \mathbf{pass} \mid \epsilon\notin\mathcal{F} \text{ and } ((\mu\in\Sigma_U, \mu\notin\mathbf{out}(\mathcal{C})) \text{ or } (\mu=\theta, \delta\notin\mathbf{out}(\mathcal{C})))\}\\ &+ \textstyle\sum\{\mu; t_\mu \mid \mu\in\Sigma_U,\ \mu\in\mathbf{out}(\mathcal{C})\}\\ &+ \textstyle\sum\{\theta; t_\theta \mid \delta\in\mathbf{out}(\mathcal{C})\}\end{aligned}$$

where t_μ and t_θ are obtained by recursively applying the algorithm for $\mathcal{C}\,\mathbf{after}\,\mu$ with $\mathcal{F}' = \{\sigma\in\Sigma_\delta^* \mid \mu\cdot\sigma\in\mathcal{F}\}$, and $\mathcal{C}\,\mathbf{after}\,\delta$ with $\mathcal{F}' = \{\sigma\in\Sigma_\delta^* \mid \delta\cdot\sigma\in\mathcal{F}\}$, respectively.

It is imperative that such an algorithm only produces test cases which are sound w.r.t. $\mathbf{ioco}_{\mathcal{F}}$ and a given specification, i.e. an implementation which is $\mathbf{ioco}_{\mathcal{F}}$-correct passes every test case generated by the algorithm. Furthermore we want completeness, i.e. for every implementation which is not $\mathbf{ioco}_{\mathcal{F}}$-correct, the algorithm can in principle generate a test case which detects such a non-conformance. The following definition formalises these properties based on a given test suite:

Definition 9. *Let $\mathcal{L}$ be a specification LTS and let T be a test suite, then for an implementation relation* $\mathbf{ioco}_{\mathcal{F}}$*:*

$$\begin{array}{lll} T \textit{ is sound and complete} & =_{\text{def}} & \forall \mathcal{P} : \mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ \mathcal{L} \Leftrightarrow \mathcal{P}\ \mathbf{passes}\ T \\ T \textit{ is sound} & =_{\text{def}} & \forall \mathcal{P} : \mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ \mathcal{L} \Rightarrow \mathcal{P}\ \mathbf{passes}\ T \\ T \textit{ is complete} & =_{\text{def}} & \forall \mathcal{P} : \mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ \mathcal{L} \Leftarrow \mathcal{P}\ \mathbf{passes}\ T \end{array}$$

Theorem 1 (Tretmans [8]). *Let $\mathcal{L}$ be an LTS and let $\mathcal{F} \subseteq Straces(\mathcal{L})$.*

1. *A test case obtained with the algorithm given in Def. 8 from $\mathcal{L}$ and $\mathcal{F}$ is sound for $\mathcal{L}$ w.r.t.* $\mathbf{ioco}_{\mathcal{F}}$.
2. *The set of all possible test cases that can be obtained with the algorithm in Def. 8 is complete.*

Remark that test cases obtained with the algorithm given in Def. 8 have finite behaviour. Nevertheless, this does not imply that they are finitely branching, i.e. a test case can specify for a possibly infinite set of outputs how to proceed next; this problem can be seen as a *state space explosion*. This makes the algorithm in general only feasible for LTSs with finite action alphabets at best.

4 Symbolic Transition Systems

While conceptually LTSs are nice, they lack the required level of abstraction for modelling complex systems. We next define the model of *Symbolic Transition Systems* (STSs). STSs extend on LTSs by incorporating an explicit notion of data and data-dependent control flow (such as guarded transitions), founded on first order logic. The STS model clearly reflects the LTS model, which is done to smoothly transfer LTS-based test theory concepts to an STS-based test theory. The model is kept as simple as possible to avoid unnecessary case distinctions in subsequent definitions and theorems.

Definition 10. *A* Symbolic Transition System *is a tuple $\langle L, l_0, \mathcal{V}, \iota, \mathcal{I}, \Lambda, \rightarrow\rangle$:*

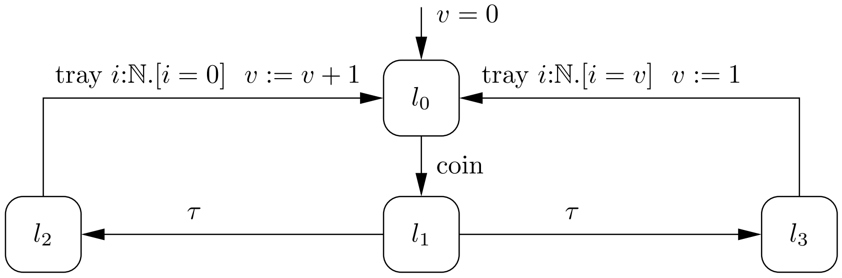
- *L is a countable set of* locations *and $l_0 \in L$ is the* initial location.
- *$\mathcal{V}$ is a countable set of* location variables.
- *$\iota \in \mathfrak{T}(\emptyset)^{\mathcal{V}}$ is an* initialisation *of the location variables.*
- *$\mathcal{I}$ is a set of* interaction variables, *disjoint from $\mathcal{V}$.*
- *Λ is a finite set of* gates. *The* unobservable gate *is denoted τ ($\tau \notin \Lambda$); we write Λ_τ for $\Lambda \cup \{\tau\}$. The* arity *of a gate $\lambda \in \Lambda_\tau$, denoted* $\mathsf{arity}(\lambda)$, *is a natural number. The* type *of a gate $\lambda \in \Lambda_\tau$, denoted* $\mathsf{type}(\lambda)$, *is a tuple of length* $\mathsf{arity}(\lambda)$ *of distinct interaction variables. We fix* $\mathsf{arity}(\tau) = 0$, *i.e. the unobservable gate has no interaction variables.*
- *$\rightarrow \subseteq L \times \Lambda_\tau \times \mathfrak{F}(\mathcal{V} \cup \mathcal{I}) \times \mathfrak{T}(\mathcal{V} \cup \mathcal{I})^{\mathcal{V}} \times L$ is the* switch relation. *We write $l \xrightarrow{\lambda,\varphi,\rho} l'$ instead of $(l, \lambda, \varphi, \rho, l') \in \rightarrow$, where φ is referred to as the* switch restriction *(acting as a guard) and ρ as the* update mapping. *We require* $\mathsf{free}(\varphi) \cup \mathsf{var}(\rho) \subseteq \mathcal{V} \cup \mathsf{type}(\lambda)$[1].

[1] Note that, here, we treat a tuple of variables as a set of variables.

In line with LTSs and IOTSs, we partition a set of gates Λ in *input gates* Λ_I and *output gates* Λ_U. Moreover, for the remainder of the paper, we consider STSs to which the following restrictions apply:

1. All sequences of τ-switches have finite length. Thus, we also do not allow for (syntactic) τ-loops.
2. For each location $l \in L$, the set of outgoing switches $\{(l, \lambda, \varphi, \rho, l') \mid l \xrightarrow{\lambda,\varphi,\rho} l'\}$ is finite, i.e. we restrict to finitely symbolic branching STSs.

Example 4. The STS $\langle \{l_0, l_1, l_2, l_3\}, l_0, \{v\}, \{v \mapsto 0\}, \{i\}, \{\text{coin}, \text{tray}\}, \rightarrow \rangle$, is depicted in Fig. 2, where $\rightarrow$ is given by the directed edges linking the locations. It models a simple slot-machine, in which a player can insert a coin, and (non-deterministically) win the jackpot (modelled by passing v coins over interaction variable i of output gate tray) or lose his coin. After that, the slot machine behaves as initially, but with a different amount of coins in the jackpot. □



$v = 0$
tray i:ℕ.$[i = 0]$ $v := v + 1$
tray i:ℕ.$[i = v]$ $v := 1$
l_0
coin
l_2
τ
l_1
τ
l_3

Fig. 2. An STS representing a simple slot-machine

We define the semantics of an STS by associating it to an LTS.

Definition 11. *Let $\mathcal{S} = \langle L, l_0, \mathcal{V}, \iota, \mathcal{I}, \Lambda, \rightarrow \rangle$ be an STS. The interpretation of $\mathcal{S}$ is given by the LTS $[\![\mathcal{S}]\!] = \langle S, s_0, \Sigma, \rightarrow \rangle$, where*

- $S = L \times \mathfrak{U}^{\mathcal{V}}$ *is the set of* states.
- $s_0 = (l_0,\ \mathsf{eval} \circ \iota) \in S$ *is the* initial *state.*
- $\Sigma = \bigcup_{\lambda \in \Lambda_\tau} (\{\lambda\} \times \mathfrak{U}^{\mathsf{arity}(\lambda)})$, *is the set of* actions.
  $\Sigma_I = \bigcup_{\lambda \in \Lambda_I} (\{\lambda\} \times \mathfrak{U}^{\mathsf{arity}(\lambda)})$, *and, analogously,* $\Sigma_U = \bigcup_{\lambda \in \Lambda_U} (\{\lambda\} \times \mathfrak{U}^{\mathsf{arity}(\lambda)})$.
- $\rightarrow \subseteq S \times \Sigma \times S$ *is the transition relation, defined by the rule of Table 4.*

In Sect. 3.1, the $\mathbf{ioco}_{\mathcal{F}}$ relation was defined as a relation between an implementation, modelled as an IOTS, and a specification, given as an LTS. We lift this definition to the level of STSs by appealing to their semantics.

Definition 12. *Let $\mathcal{S}$ be an STS and $\mathcal{P}$ a physical system, modelled as an IOTS. Then $\mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ \mathcal{S}$ iff $\mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ [\![\mathcal{S}]\!]$.*

Table 4. Deduction rule for transitions

$$\frac{l \xrightarrow{\lambda,\varphi,\rho} l' \quad \mathsf{type}(\lambda) = \langle \nu_1, \ldots, \nu_n \rangle \quad \varsigma \in \mathfrak{U}^{\mathsf{type}(\lambda)} \quad \vartheta \cup \varsigma \models \varphi \quad \vartheta' = (\vartheta \cup \varsigma)_{\mathsf{eval}} \circ \rho}{(l, \vartheta) \xrightarrow{(\lambda, \langle \varsigma(\nu_1), \ldots, \varsigma(\nu_n) \rangle)} (l', \vartheta')}$$

5 On-the-Fly Testing

Lifting the $\mathbf{ioco}_{\mathcal{F}}$ test theory to STSs by appealing to their semantics, as we did in the previous section, puts us in a position to reuse the standard algorithm of Sect. 3.2 for STSs. However, as we already remarked in that section, that algorithm suffers from a state space explosion. Note that also the computation of the LTS that is associated to an STS in general is of infinite size.

5.1 Symbolic Ingredients

Given an STS with a switch relation $\rightarrow$. We define a generalised switch relation $\Longrightarrow \subseteq L \times \Lambda_\tau \times \mathfrak{F}(\mathcal{V} \cup \mathcal{I}) \times \mathfrak{T}(\mathcal{V} \cup \mathcal{I})^{\mathcal{V}} \times L$ (see the deduction rules of Table 5). The intuition behind this relation is that it abstracts from the unobservable events that possibly precede and follow an observable event. It is subsequently used in the definition of a symbolic counterpart of the **after** relation of Sect. 3.1.

Table 5. Deduction rules for generalised switches

$$l \xrightarrow[]{\tau,\top,\mathsf{id}}\!\!\!\!\!\!\Longrightarrow l \qquad \frac{l \overset{\tau,\varphi,\rho}{\Longrightarrow} l''' \quad l''' \xrightarrow{\lambda,\psi,\pi} l'' \quad l'' \overset{\tau,\chi,\zeta}{\Longrightarrow} l' \quad \lambda \in \Lambda_\tau}{l \overset{\lambda,\, \varphi \wedge \psi[\rho] \wedge (\chi[\pi])[\rho],\, [\rho] \circ [\pi] \circ \zeta}{\Longrightarrow} l'}$$

Definition 13. *Let $\langle L, l_0, \mathcal{V}, \iota, \mathcal{I}, \Lambda, \rightarrow \rangle$ be an STS.*

- *An* instantiated location *is a pair (l, ϖ), where $l \in L$ is a location and ϖ is a mapping of the set of location variables to ground terms, i.e. $\varpi \in \mathfrak{T}(\emptyset)^{\mathcal{V}}$.*
- *A* stimulus *(resp.* reaction*) is a pair (λ, η), where $\lambda \in \Lambda_I$ is an input gate (resp. $\lambda \in \Lambda_U$ is an output gate) and $\eta \in \mathfrak{T}(\emptyset)^{\mathsf{type}(\lambda)}$ is a mapping of the interaction variables of λ to ground terms.*

Input constraints represent the conditions for the input gates under which an instantiated location is specified to proceed.

Definition 14. *Let (l, ϖ) be an instantiated location. The* input constraints *for (l, ϖ), denoted $\Omega(l, \varpi)$, are defined as*

$$\Omega(l, \varpi) = \bigcup_{\lambda \in \Lambda_I} \{(\lambda, \bigvee \{\psi[\varpi] \mid l \xrightarrow{\lambda,\psi,\rho} l'\})\}$$

We generalise this to $\Omega(\mathcal{C}) = \bigcup_{(l,\varpi) \in \mathcal{C}} \Omega(l, \varpi)$.

The concept of quiescence (cf. Sect. 3.1) is lifted to the level of STSs.

Definition 15. *An instantiated location* (l, ϖ) *is* quiescent*, denoted* $\delta(l, \varpi)$*, iff:*

$$\forall \lambda \in \Lambda_U \cup \{\tau\} : \neg\big(\exists l' : l \xrightarrow{\lambda,\varphi,\rho} l' \text{ with } \mathfrak{M} \models \overline{\exists}(\varphi[\varpi])\big) \tag{5}$$

By observing a reaction or providing a stimulus (λ, η) at an instantiated location (l, ϖ), location l is left and some location of a set of new locations (with updated location variables) can be reached. This set is given by the operator $\mathbf{after}_s$:

$$(l, \varpi)\, \mathbf{after}_s(\lambda, \eta) = \{(l', [\eta] \circ [\varpi] \circ \pi) \mid l \xRightarrow{\lambda,\psi,\pi} l' \text{ and } \mathfrak{M} \models (\psi[\varpi])[\eta]\} \tag{6}$$

For the special case where quiescence is observed, we define:

$$(l, \varpi)\, \mathbf{after}_s\, \delta = \{(l', [\varpi] \circ \pi) \mid l \xRightarrow{\tau,\psi,\pi} l', \mathfrak{M} \models \psi[\varpi] \text{ and } \delta(l', [\varpi] \circ \pi)\} \tag{7}$$

We overload the operator $\mathbf{after}_s$ to yield the set of instantiated locations that are reached when the stimulus or reaction is made from a given set of instantiated locations. Let $\mathcal{C} \subseteq L \times \mathfrak{T}(\emptyset)^{\mathcal{V}}$ and x be a stimulus or reaction, including quiescence. Then $\mathcal{C}\, \mathbf{after}_s\, x = \bigcup_{(l,\varpi)\in\mathcal{C}} (l, \varpi)\, \mathbf{after}_s\, x$.

5.2 Algorithm

To avoid the state space explosion problem, we combine test generation from STSs with an on-the-fly execution of the test cases. This means that the generation of the test case proceeds in lock-step with its execution, see also [1]. This has the advantage, that only the part of the state space is generated, which corresponds to the observations made while testing.

To implement the test generation for the $\mathbf{ioco}_{\mathcal{F}}$ relation we assume that there is a function $\texttt{InF}$:$\Sigma_\delta^* \to \texttt{boolean}$ to decide whether the currently executed (suspension) trace is an element of $\mathcal{F}$, i.e. $\texttt{InF}(\sigma) = \texttt{true} \Leftrightarrow \sigma \in \mathcal{F}$. The algorithm keeps track of the executed trace σ and checks if $\texttt{InF}(\sigma)$ holds before giving verdicts. In the case of $\mathbf{ioco}_{Straces(\mathcal{L})}$ (which is implemented in the test tool TorX [9]), $\texttt{InF}(\sigma) = \texttt{true}$ for all σ, and can therefore be omitted in the algorithm.

The algorithm we present next follows the same structure as the one in Sect. 3.2. It maintains a set of instantiated locations $\mathcal{C}$ which symbolically represents the set of states in which the SUT may currently be. This is in general not a singleton (due to possible non-determinism in system specifications), but it is always finite. This is because we restrict to STSs which are finitely branching, and which do not allow for infinite sequences of τ-switches. Furthermore, all these locations in $\mathcal{C}$ are instantiated due to an on-the-fly execution, i.e. the algorithm knows for every location the actual values of the location variables. We first present the algorithm, and subsequently discuss it.

Definition 16. *Given an STS* $\mathcal{S} = \langle L, l_0, \mathcal{V}, \iota, \mathcal{I}, \Lambda, \rightarrow\rangle$ *and an SUT. Let* $\mathcal{C}$ *be a non-empty set of instantiated locations and let* σ *be a suspension trace of* $[\![\mathcal{S}]\!]$*. Initially, we use* $\mathcal{C} = \{(l, \rho[\iota]) \mid l_0 \xRightarrow{\tau,\varphi,\rho} l, \text{ with } \mathfrak{M} \models \varphi[\iota]\}$ *and* $\sigma = \epsilon$*. The algorithm executes a finite number of applications of the following three non-deterministic choices:*

```
(1) Stop testing
    01. Give the verdict pass.
(2) Give input to the SUT
    02. Compute $\Omega(\mathcal{C})$.
    03. Choose $(\lambda,\psi)\in\Omega(\mathcal{C})$ and a stimulus $(\lambda,\eta)$, such that $\mathfrak{M} \models \psi[\eta]$.
    04. Send $w=\langle \mathsf{eval}(\eta(\nu_1)),\ldots,\mathsf{eval}(\eta(\nu_n))\rangle$ over $\lambda$, where $\langle\nu_1,\ldots,\nu_n\rangle=\mathsf{type}(\lambda)$.
    05. Compute $\mathcal{C}' = \mathcal{C}\,\mathbf{after}_s(\lambda,\eta)$.
    06. Repeat the algorithm with the set $\mathcal{C}'$ and trace $\sigma' = \sigma\cdot(\lambda,w)$.
(3) Observe output of the SUT
    07. If quiescence is observed then
    08.     Compute $\mathcal{C}' = \mathcal{C}\,\mathbf{after}_s\,\delta$.
    09.     If $\mathcal{C}' \neq \emptyset$ then
    10.       Repeat the algorithm with set $\mathcal{C}'$ and trace $\sigma' = \sigma\cdot\delta$.
    11.     else
    12.       Give verdict fail when InF($\sigma$), and pass otherwise.
    13. else
    14.     Receive $w = \langle w_1,\ldots,w_n\rangle$ over $\lambda$.
    15.     Compute $\eta$, satisfying $\mathsf{eval}(\eta(\nu_i)) = w_i$ for all $\nu_i\in\mathsf{type}(\lambda)$.
    16.     Compute $\mathcal{C}' = \mathcal{C}\,\mathbf{after}_s(\lambda,\eta)$.
    17.     If $\mathcal{C}' \neq \emptyset$ then
    18.       Repeat the algorithm with set $\mathcal{C}'$ and trace $\sigma' = \sigma\cdot(\lambda,w)$.
    19.     else
    20.       Give verdict fail when InF($\sigma$), and pass otherwise.
```

The above algorithm shares the base case *(1)* with the algorithm of Def. 8: it can terminate at any moment and give the verdict **pass**.

Differently from the algorithm of Def. 8, before sending an input to the SUT (in case *(2)*), first a set of input constraints for $\mathcal{C}$ is computed (line 02). This is a set of first order formulas specifying under which conditions certain data can be sent over one of the input gates. The input constraints in fact represent a subset of the possibly infinite set of inputs. The input constraint and the stimulus that are subsequently chosen in line 03 serve to identify an appropriate input w, which is sent over gate λ in line 04. The algorithm then proceeds with the calculation of a new set of instantiated locations (line 05), sets the new suspension trace, and continues with these new parameters, line 06.

When observing quiescence of the SUT (case *(3)*, line 07), we first check whether this is actually specified behaviour (lines 08 – 10) or not (lines 11 – 12). In the first case, the algorithm continues with the newly obtained set of instantiated locations and suspension trace. In the latter case, we assign the verdict **fail** when the executed trace was an element of $\mathcal{F}$, and **pass** otherwise.

If the SUT actually produces an output (case *(3)*, line 14), we receive a data value w over an output gate λ. To facilitate reasoning about this data value, we first find a corresponding mapping to ground terms η (line 15). Note that this η represents the *actual, concrete* values that are passed over the gate λ. Next, in line 16, the new set of instantiated locations found after observing reaction (λ,η), is computed. Note that since η represents the concrete values for the interaction variables, and due to the restrictions we pose on STSs, this new set of instantiated locations is finite. In line 17, it is tested whether the observed output was allowed, and if so, testing is continued with the new set in line 18. When the observed output is not allowed (line 19), we assign the verdict **fail** or **pass**, dependent on whether the trace we executed thus far was part of $\mathcal{F}$. Note that the meaning of **pass** in lines 12 and 20 corresponds more to an **inconclusive** verdict (see also [7]). However, this verdict is currently not part of our test case definition.

Next we state the correctness and completeness of the algorithm above. That means that we have not lost any detection power compared to the (infeasible) algorithm of Sect. 3.2.

Theorem 2. *Let $\mathcal{S}$ be an STS and let $\mathcal{F} \subseteq$ Straces$(\llbracket\mathcal{S}\rrbracket)$. Given an SUT assumed to behave like an IOTS $\mathcal{P}$ we have:*

1. *$\mathcal{P}\,\mathbf{ioco}_{\mathcal{F}}\,\mathcal{S} \Rightarrow$ every application of the algorithm given in Def. 16 on $\mathcal{S}, \mathcal{F}$ and the SUT results in* **pass**.
2. *$\neg(\mathcal{P}\,\mathbf{ioco}_{\mathcal{F}}\,\mathcal{S}) \Rightarrow$ there exists an application of the algorithm given in Def. 16 on $\mathcal{S}, \mathcal{F}$ and the SUT which potentially results in* **fail**.

The *potentially* in *2.* is because the SUT can behave non-deterministically: if the SUT chooses (non-deterministically) a non-erroneous path, the algorithm cannot observe the fault, of course.

5.3 Discussion

The decidability (and computability) of the first order formulas occurring in STSs is an issue of utmost importance when considering a computer implementation of the algorithm of Def. 16. Two entities, viz. the set of input constraints $\Omega(\mathcal{C})$ and partly the new sets of instantiated locations $\mathcal{C}\,\mathbf{after}_s(\lambda, \eta)$ can be computed purely on the basis of syntax. At some point, though, it is necessary to decide whether a (possibly existentially closed) formula has a solution. In general, this may not even be computable. While we did not address this issue in this paper, as it is orthogonal to the general idea behind the algorithm we presented, we did identify where decidability and computability are of concern. A way to proceed here is to use feasible subsets of first order logic, possibly assisted by (dedicated) theorem provers.

A second point of attention is the selection of appropriate stimuli to be passed on to the SUT (case *(2)* of the algorithm). While the question of decidability and computability is certainly important here, the strategy of filtering interesting stimuli out of a huge set of mainly uninteresting input stimuli satisfying some constraint in the set $\Omega(\mathcal{C})$ is equally challenging. This is where tools such as GAST may come into play. Such tools can automatically generate such stimuli based on given strategies. For instance, GAST uses generics to represent a data type; using a strategy which is similar to unfolding and traversing a tree-like structure, values of the data type are obtained. Other strategies are to employ the syntactical structure of a data type, or to use some uniformity hypothesis for generating and selecting interesting data values.

6 Conclusions

We have tackled the state space explosion problem that is often encountered in state-based test tools. This is achieved by lifting a test theory for Labelled Transition Systems (LTSs), called $\mathbf{ioco}_{\mathcal{F}}$, to Symbolic Transition Systems (STSs). Unlike in LTSs, data is treated symbolically in an STS. As a side-effect, system descriptions given as an STS are at a natural level of abstraction and in general more concise than their LTS counterparts. In fact, the semantics of STSs (which is given by a translation to LTSs) can yield LTSs of infinite size.

Due to this LTS semantics of the STS, the original $\mathbf{ioco}_{\mathcal{F}}$ test relation could be reused in our symbolic setting, including the classical test case generation algorithm for $\mathbf{ioco}_{\mathcal{F}}$. While in theory, this algorithm generates test cases that can be infinitely branching, in practice, this is effectively solved by an on-the-fly implementation of the algorithm working directly on STSs. This solution is only apparent on account of the orthogonal treatment of data and control in STSs.

Several issues remain open, such as the identification of feasible subsets of first order formulas and a running implementation of our algorithm.

References

1. A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.
2. A. Belinfante, L. Frantzen, and C. Schallhart. Tools for test case generation. In M. Broy, B. Jonsson, J.P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-based Testing of Reactive Systems - A Seminar Volume*, LNCS. Springer Verlag, 2004. To appear.
3. K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, 2000.
4. M.R.A. Huth and M. Ryan. *Logic in computer science: modelling and reasoning about systems.* Cambridge University Press, 2000.
5. P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In *Proceedings 14th International Workshop on the Implementation of Functional Languages, IFL 2002, Selected Papers, Madrid, Spain, September 16-18, 2002, Springer Verlag, LNCS 2670*, pages 84–100, 2003.
6. P. Koopman and R. Plasmeijer. Testing reactive systems with GAST. In *Proceedings Fourth symposium on Trends in Functional Programming, Edinburgh, Scotland, September 11-12, 2003.*, 2004.
7. V. Rusu, L. du Bousquet, and T. Jéron. An Approach to Symbolic Test Generation. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods – IFM 2000*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer-Verlag, 2000.
8. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.
9. J. Tretmans and E. Brinksma. TorX : Automated Model Based Testing. In A. Hartman and K. Dussa-Zieger, editors, *First European Conference on Model-Driven Software Engineering.* Imbuss, Möhrendorf, Germany, December 11-12 2003.